590665

Leap Before You Look:
Information Gathering in the PUCCINI planner

Keith Golden
NASA Ames Research Center
M/S 269-2
Moffett Field, CA 94035-1000
kgolden@ptolemy.arc.nasa.gov
(650) 604-3585

Abstract

Most of the work in planning with incomplete information takes a "look before you leap" perspective: Actions must be guaranteed to have their intended effects before they can be executed. We argue that this approach is impossible to follow in many real-world domains. The agent may not have enough information to ensure that an action will have a given effect in advance of executing it. This paper describes PUCCINI, a partial-order planner used to control the Internet Softbot (Etzioni & Weld 1994). PUCCINI takes a different approach to coping with incomplete information: "Leap before you look!" PUCCINI doesn't require actions to be known to have the desired effects before execution. However, it still maintains soundness, by requiring the effects to be verified eventually. We discuss how this is achieved using a simple generalization of causal links.

Introduction

A boy's appetite grows very fast, and in a few moments the queer, empty feeling had become hunger, and the hunger grew bigger and bigger, until soon he was as ravenous as a bear.

Poor Pinocchio ran to the fireplace where the pot was boiling and stretched out his hand to take the cover off, but to his amazement the pot was only painted! Think how he felt! His long nose became at least two inches longer.

He ran about the room, dug in all the boxes and drawers, and even looked under the bed in search of a piece of bread, hard though it might be, or a cookie, or perhaps a bit of fish. A bone left by a dog would have tasted good to him! But he found nothing. ...

Suddenly, he saw, among the sweepings in a corner, something round and white that looked very much like a hen's egg. In a jiffy he pounced upon it. It was an egg.

— Carlo Collodi,[1] *The Adventures of Pinocchio*

Pinocchio's search for food is evocative, in part, because it is so familiar. Whether looking for food, a passport, or information on the Web, we have all had the experience of searching exhaustively for something until we find it. Even though any single action, such as opening a drawer or looking under the bed, is likely to result in failure, we know that if we search long enough, we are likely to find what we're looking for. Pinocchio's attempt to lift the cover from the pot is also familiar, since attempts to open locked doors or copy read-protected files result in similar frustration of our expectations. What these activities have in common is some precondition that we *assumed* to be true, but later found to be false.

We are interested in the problem of building agents that can solve user goals in software environments, such as the Unix operating system or World Wide Web, in which the agent has massively incomplete (but correct) information about the world. One such agent is the Internet Softbot (Etzioni & Weld 1994). Internet resources and Unix commands are represented as planner actions, and a planner, called PUCCINI,[2] is used to find some combination of these actions that together will achieve the user's goal.

It should not be surprising to anyone who has looked for something on the Web that agents in such environments could spend much of their time searching for files or Web pages in much the way that Pinocchio searched for something to eat. However, most planners that deal with incomplete information don't behave much like Pinocchio.

Most planners, by adopting some form of *knowledge preconditions* (Moore 1985), require the agent to know, *a priori*, that an action will have some desired result. As we argued in (Golden & Weld 1996), and will briefly discuss here, these knowledge preconditions are representational handcuffs, which make action representations more awkward and limit the utility of our planners; our action language, SADL, eliminates them. In this paper, we show how a simple generalization of *causal links* allows a planner to exploit the elimination of knowledge preconditions, without giving up soundness. We show empirically that this added expressiveness does not degrade planner performance.

The remainder of the paper is organized as follows.

[1] Translated from the Italian by Carol Della Chiesa

[2] PUCCINI stands for Planning with Universal quantification, Conditional effects, Causal links, and INcomplete Information. PUCCINI is a partial-order planner based on UCPOP (Penberthy & Weld 1992). An earlier version of PUCCINI was called XII.

First, we introduce the fundamentals of the SADL language and the PUCCINI planner. Then, in the next section, we briefly discuss the problem with knowledge preconditions. Although knowledge preconditions, as inflexible requirements of the planner, are harmful, it is still necessary for the planner to gather information in support of planning. In the following section, we show how that is done in PUCCINI. Then we consider the option of *assuming* that certain preconditions hold, performing the action, and verifying the preconditions afterward. Finally, we evaluate the cost of this added flexibility.

Back in the SADL

PUCCINI goals and actions are described using the language SADL,[3] which builds on UWL (Etzioni *et al.* 1992) and ADL (Penberthy 1993). Like UWL, SADL is designed to represent sensing actions and information goals. To distinguish sensory effects from causal effects and goals of information from traditional goals of satisfaction, SADL provides *annotations* for goals and effects.

Following UWL, SADL divides effects into those that change the world, annotated by **cause**, and those that merely report on the state of the world, annotated by **observe**. Executing actions with **observe** effects assigns values to *runtime* variables that appear in those effects. By using a runtime variable (syntactically identified with a leading exclamation point, *e.g.* *!tv*) as a parameter to a later action (or to control contingent execution), information gathered by one action can affect the agent's subsequent behavior. For example, `ping twain` has the effect of **observe** (machine.alive(`twain`), *!tv*), *i.e.* determining whether it is true or false that the machine named `twain` is alive, and `wc myfile` has the effect **observe** (word.count(`myfile`, *!word*)), *i.e.* determining the number of words in `myfile`. The variable *!tv*, above, is the truth value of the proposition machine.alive(`twain`). All literals have truth values expressed in a three-valued logic: `T`, `F`, `U` (unknown), or represented by a variable. If a truth value is not specified, it defaults to `T`.

Goals are similarly annotated. The goal **satisfy**(P) indicates a traditional goal (as in ADL): achieve P by whatever means possible. In the presence of incomplete information, we make the further requirement that the agent knows that P is true, so **satisfy**(P) means that KNOW(P) must be true in the final state of the plan. Free variables are implicitly existentially quantified, and the quantifier takes the widest possible scope. For example, **satisfy**(in.dir (f, `tex`), `T`) means "Ensure that there's at least one file in directory `tex`," and **satisfy**(in.dir (`myfile`, `tex`), *tv*) means "Find out whether or not `myfile` is in `tex`."

The **initially** annotation, introduced in (Golden & Weld 1996), is similar to **satisfy**, but it refers to the time when the goal is given to the agent, not to the time when the goal is achieved. **initially**(P, *tv*) means that by the time the agent has finished executing the plan, it should know whether P was true when it started. **initially**(P) is not achievable by an action that changes the fluent P, since such an action only obscures the initial value of P. However, changing P after determining its initial value is fine. By combining **initially** with **satisfy** we can express "tidiness" goals: modify P at will, but restore its initial value by plan's end (Golden & Weld 1996; Weld & Etzioni 1994). Furthermore, we can express goals such as "Find the the file currently named `paper.tex`, and rename it to `kr.tex`," which is beyond the expressive power of most planners (Golden & Weld 1996).

The **hands-off** annotation indicates a maintenance goal that prohibits the agent from changing the fluent in question.

Like ADL, SADL also supports universal quantification and conditional effects. Combining these features with **observe** effects yields expressive sensor models, such as those shown in the next section.

PUCCINI overview

PUCCINI is a partial-order planner in the same family as SNLP (McAllester & Rosenblitt 1991) and UCPOP (Penberthy & Weld 1992). It builds plans incrementally by starting with an empty plan and a *goal agenda* of goals that need to be achieved. When goals are achieved, they are removed from the goal agenda. When actions are added to the plan in support of goals, their preconditions are added to the goal agenda. This process continues until the goal agenda is empty or some goal proves impossible to achieve. When a goal has been "achieved" by adding an action to the plan, we must guard against the possibility that some other action added later will "clobber" the goal, forcing the planner to re-achieve it. To prevent this from happening, the planner adds a *causal link* (Tate 1977), which records its commitment to achieve a given goal by using the effect of a given action. If effect e of A_1 is used to support precondition q of A_2, we represent the corresponding causal link as $A_1 \xrightarrow{e,q} A_2$. A_1 is required to precede A_2 so that q will be achieved by the time it's needed. Any change to the plan that would violate the causal link is called a *threat* to the link, and must be resolved by the planner. For example, an action A_t with the effect $\neg q$ possibly occurring between actions A_1 and A_2 would threaten the link $A_1 \xrightarrow{e,q} A_2$. This threat might be resolved by adding an ordering constraint to ensure that A_t is executed before A_1 or after A_2. In addition to achieving goals on the goal agenda and resolving threats, PUCCINI must execute actions. These three procedures comprise the top-level PUCCINI algorithm (Figure 1). This paper only focuses on a narrow aspect of the goal achievement procedure (Figure 6). For a discussion of the other aspects of the algorithm, consult (Golden 1997; Etzioni, Golden, & Weld 1997).

Knowledge Preconditions

Knowledge preconditions are meant to capture the information needed by an agent to execute an action for a given purpose. For example, an agent opening a safe

[3] SADL stands for Sensory Action Description Language.

PUCCINI $(\langle \mathcal{A}, \mathcal{O}, \mathcal{B}, \mathcal{C}, \mathcal{E} \rangle, \mathcal{G}, \mathcal{D}, s)$

1. **If $\mathcal{G} = \emptyset \wedge \mathcal{E} = \emptyset \wedge \forall \ell \in \mathcal{C}$ ℓ not threatened, return** success.
2. **Pick one of**
   - **(a) HandleGoal($\mathcal{A}$, $\mathcal{O}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{G}$, $\mathcal{D}$)**
   - (b) HandleThreats($\mathcal{O}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{G}$)
   - (c) s := HandleExecution($\mathcal{A}$, $\mathcal{O}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{E}$, s)
3. **PUCCINI($\langle \mathcal{A}, \mathcal{O}, \mathcal{B}, \mathcal{C}, \mathcal{E} \rangle$, $\mathcal{G}$, $\mathcal{D}$, s)**

Figure 1: The PUCCINI Algorithm takes as input a plan, a goal agenda ($\mathcal{G}$), a domain theory ($\mathcal{D}$), and the current state of the world, s, which is only partially known. The plan consists of a set of actions ($\mathcal{A}$), ordering relations on $\mathcal{A}$ ($\mathcal{O}$), variable binding constraints ($\mathcal{B}$), causal links ($\mathcal{C}$) and unexecuted actions in $\mathcal{A}$ ($\mathcal{E}$). The planner repeatedly fixes "flaws" in the plan (open goals, threats and unexecuted actions) until the plan is complete. The lines of the algorithm relevant to this paper are shown in **bold**.

needs to know the combination. In (Golden & Weld 1996), we argued that the practice of specifying knowledge preconditions for actions was too restrictive and should be abandoned.

Moore (Moore 1985) identified two kinds of knowledge preconditions an agent must satisfy in order to execute an action in support of some proposition P: First, the agent must know a rigid designator (*i.e.*, an unambiguous, executable description) of the action. Second, the agent must know that executing the action will in fact achieve P. Subsequent work by Morgenstern (Morgenstern 1987) generalized this framework to handle scenarios where multiple agents reasoned about each other's knowledge.

The first type of knowledge precondition doesn't present any problem for us, since, in our language, *all* actions are rigid designators. `dial`(combination(`safe`)) is not an admissible action, but `dial(31-24-15)` is. Lifted action schemas, *e.g.* `dial(`x`)`, are not rigid designators, but it is easy to produce one by substituting a constant for x.

Moore's second type of knowledge precondition presupposes that an action in a plan must provably succeed in achieving a desired goal. This is a standard assumption in classical planning, but is overly restrictive given incomplete information about the world; enforcing this assumption by adding knowledge preconditions to actions is inappropriate. For example, if knowledge of the safe's combination is a precondition of the `dial` action, then it becomes impossible for a planner to solve the goal "find out whether the combination is 31-24-15" by dialing that number, since before executing the `dial` action, it will need to satisfy that action's precondition of finding out whether 31-24-15 is the right combination!

Eliminating the knowledge precondition from the `dial` action also allows the unhurried agent to devise a plan to enumerate the possible combinations until it finds one that works.[4] While this may seem silly, Pinocchio was following an identical strategy in his hunt for food. The Internet Softbot does the same when directed to find a particular user, file or web page, whose location is unknown. If `finger` and `ls` (Figures 2 and 3) included knowledge preconditions, then the actions would be useless for locating users and files. For example, `ls /papers` only returns information about the file `aips.tex` if `aips.tex` is in `/papers`. Yet planning to move `aips.tex` into `/papers` misses the point if the goal is to *find* `aips.tex`!

In a broad class of domains, which we call *knowledge-free Markov* (KFM) domains, the effects of an action depend only on the state of the world (and *not* on the agent's knowledge about the world) at the time of execution. In such domains, actions are best encoded without knowledge preconditions. Simple mechanical and software systems are naturally encoded as KFM, while domains involving abstract actions are typically *not*. Such actions represent complex (albeit sketchy) plans in their own right, and depend on the agent's knowledge to be executed successfully.

Although knowledge *preconditions* are problematic, it is often useful for an agent to plan to obtain information, such as the combination of a safe, either to reduce search or to avoid dangerous mistakes. For example, if there's an alarm on the safe, then it would be a bad idea to try all combinations. More importantly, it is necessary that the agent know, by the time it is finished, that it has achieved the goal. If we don't maintain this constraint, our planner is not even sound!

Planning to Sense

The solution is to give the planner the information needed to determine when obtaining information, such as the combination of a safe, would be useful, and then leave it to the planner to decide how and when to acquire that information. We do so quite simply by the use of conditional effects (see Figures 2 and 3). Following (Pednault 1986), we call the precondition of a conditional effect a *secondary precondition* (the precondition of the action itself is known as a *primary precondition*). If the agent wants to know whether a conditional effect will occur, it needs to know whether the corresponding secondary precondition is true. However, we don't require the planner to achieve the secondary preconditions before executing an action, even if it wants the corresponding effects to occur. In this sense, the secondary preconditions are *descriptive*, not *prescriptive*. The planner has the option of ensuring that these secondary preconditions are true before it executes the action, but it can also verify them after the fact. We consider the former case in this section. We will discuss the latter in the next section.

The planner can ensure that a precondition is true by either observing it to be true or making it true. For example, suppose a softbot wants to compress the file

[4] Richard Feynman estimated that he could open a safe using this method in four hours (Feynman 1985).

```
action finger(s)
    precond:  satisfy (current.shell(csh))
    effect:   ∀ !p ∃ !l, !f, !u
                when lastname(!p, s) ∨
                     firstname(!p, s) ∨
                     userid(!p, s)
                  observe (firstname(!p, !f)) ∧
                  observe (lastname(!p, !l)) ∧
                  observe (userid(!p, !u))
```

Figure 2: **Unix action schema.** A simplified version of the PUCCINI `finger` action to find information about a user. This action returns information about all users whose first name, last name or userid is equal to the input string s. Variables like $!p$, beginning with an exclamation point, are called *run-time* variables. The values of these variables are determined at run time as a result of sensing. Effects labeled with **observe** designate propositions that are sensed by the agent, as opposed to being affected.

```
action ls(d)
    precond:  satisfy (current.shell(csh))
    effect:   ∀ !f when in.dir(!f, d)
                ∃ !p, !n
                  observe (in.dir(!f, d)) ∧
                  observe (pathname(!f, !p)) ∧
                  observe (name(!f, !n))
```

Figure 3: **Unix action schema.** A simplified version of the PUCCINI `ls` action (Unix `ls -a`) to list all files in a directory. The relation in.dir($!f$, d) means file $!f$ is in directory $!d$, so this action returns information about all files in a given directory.

`aips.tex`, and decides to do so by executing the action `compress /papers/*`, which compresses every file in the directory `/papers`. The action will only succeed if `aips.tex` is actually in directory `/papers`. The softbot could subgoal on ensuring that `aips.tex` is in `/papers`, either by observing that `aips.tex` is in `/papers`, using the action `ls` (see Figure 3), or by moving `aips.tex` into `/papers`. We use the term *observational link* to denote links, $A_1 \xrightarrow{e:q} A_2$, in which the effect e is an **observe** effect.

Suppose a softbot wants to find the userid of Oren Etzioni, a University of Washington professor. It can do so using the `finger` command (Figure 2). `finger` takes a single argument, a string, and will only provide information about Oren if that string is Oren's first name, last name or userid. Since the softbot doesn't know Oren's userid, that will be useless to subgoal on, but the softbot does know Oren's first and last names. Suppose it adopts the subgoal of knowing Oren's last name. This can be satisfied using the softbot's prior knowledge about the world. The softbot's prior knowledge is represented in the planner as the effects of a dummy "initial step," A_0. So the planner adds a link from A_0 to `finger`, representing its commitment to use its knowledge of the initial state to satisfy its goal of knowing Oren's last name.

Now suppose the softbot is given the goal of finding a user with the userid `map`, *i.e.*, **initially**(userid(p, `map`)), but it has no knowledge about any users, including whatever user, if any, has the userid of `map`. The softbot could plan to execute `finger map`, but it's less obvious what to do with the secondary preconditions of `finger`. Since the softbot doesn't know anything about the user in question, it can't know that user's first name or last name. It also doesn't know what user p satisfies the relation userid(p, `map`). While we could try simply asserting that there's a user whose userid is `map`, this fact is not necessarily true, and asserting it into the softbot's knowledge base would introduce a number of complications, not the least of which is that the softbot would erroneously believe that it already knew the answer to the query.

Leap before you look

Impasses like the one above are all too common. Fortunately, they have a simple solution. In the case of `finger map`, above, the softbot can just execute the `finger`; it will know afterward what user has that userid, since the action has the effect

when userid($!p$, s) ... **observe** (userid($!p$, $!u$)).[5]

In short, the softbot will know *after* executing `finger map` whether the secondary precondition was true before executing it. Since the softbot can verify after the fact that the precondition was true, there's no reason not to go ahead and execute the action.

What we want is the ability to temporarily *assume* the secondary precondition is true, and to later *verify* that the assumption was valid by performing an observation. Formally, a commitment to verify an assumption is a quadruple $\langle A_p, p, e, A_e \rangle$, where p is a precondition of some effect of A_p. p is assumed to be true, and is to be verified by effect e of action A_e. Note that, unlike our discussion in the previous section, it is essential that the verification be done by an *observation*, since that is the only way to obtain information about the past. Executing an action that *caused* the precondition to be true would be useless, since the precondition needs to have been true *before* the action was executed. Because the observation will only be valid if the condition p remains unperturbed, we must protect p over the interval between A_p and A_e.

There is a striking similarity between these commitments to verify preconditions and observational links. In fact, they are identical to observational links, with the exception that the order of the producer and consumer is reversed! We call these commitments *verification links*, and write them as $A_p \xleftarrow{p,e} A_e$. Because we want to consider supporting these preconditions by *either* prior observation *or* later verification, we can accomplish this feat quite simply by omitting the ordering constraint that would normally be placed between the producer and consumer. Since the only difference between an observation link and a verification link is

[5] This reasoning depends on the fact that every user has at most one userid, but the planner has access to this fact.

in the ordering constraint, omitting the ordering constraint means the planner hasn't committed to which kind of link it is. Eventually, the actions will be ordered, either in the course of planning or prior to execution. Once that happens, the link will be either an observation link or a verification link, depending on the action order. Relaxing the ordering constraint allows for self-links as well, as in the case of `finger map`, above. For example, consider the following effect of `ls`:

$\forall$ *!f* **when** in.dir(*!f*, *d*) **observe** (in.dir(*!f*, *d*)),

where in.dir(*!f*, *d*) means that file *!f* is in directory *!d*. We can satisfy the in.dir precondition by linking to the effect of the same action (see Figure 4). If the desired file is not in the directory, the **observe** effect ensures that the agent will know that fact after executing the action,[6] and the assumption will be proven false.

One potential concern about verification links is that they increase the size of the search space by giving the planner more ordering options. In fact, this is inevitable, since more plans are admissible when verification links are allowed. However, the number of plans that are solutions also increases, and, due to the least-commitment approach, the alternative ordering options may never be explicitly explored. Thus, it is possible that using verification links would actually decrease the number of plans explored, at least in some cases. Whether more or fewer plans are explored is an empirical question, which we investigate in the next section.

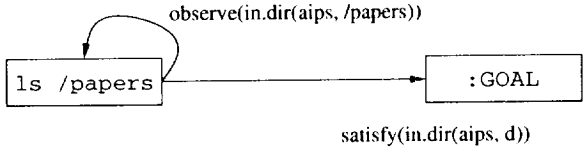

Figure 4: PUCCINI adds `ls /papers` in support of the goal of finding the file `aips.tex`. Since this relies on the conditional effect of `ls`, the desired outcome will only occur if in.dir(`aips.tex`, `/papers`) is true when `ls` is executed. Rather than trying to achieve this precondition, PUCCINI adds a verification link from the effect of `ls`, **observe** (in.dir(*!f*, `/papers`)), to this precondition. The agent will know after executing the `ls` whether the precondition was true.

Bookkeeping

While we can handle assumptions elegantly by lifting the ordering constraints imposed along with observational links, that doesn't free us from bookkeeping. The effects supported by assumptions are still contingent, and we must exercise care in what we do with them. We should not store them in the agent's knowledge base or execute actions with primary preconditions supported by them until they have been verified.

[6]Actually, it is the fact that the agent observes *all* files that enables the agent to conclude that other files are not in the directory. See (Etzioni, Golden, & Weld 1997) for a discussion of how this inference works.

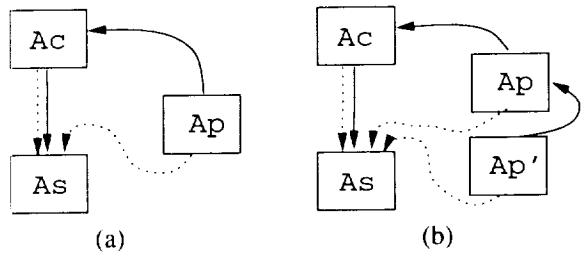

Figure 5: Bookkeeping for verification links: (a) A_p may follow A_c, but is used to verify a precondition of an effect of A_c. This effect, in turn, satisfies a primary precondition of A_s (solid lines indicate links). The effects of A_s are undefined unless the precondition is satisfied, and it won't be known whether A_c had the desired effect until A_p has been executed, so an ordering constraint is added to ensure that A_s is not executed before A_p (dotted lines indicate ordering constraints). (b) The secondary precondition of A_p itself is supported by an action, $A_{p'}$, that may be executed later. Since $A_{p'}$ is indirectly providing support for A_s, it must also be executed before A_s.

This bookkeeping is really quite simple. If a link $A_p \stackrel{q,e}{\rightarrow} A_c$ may represent a verification link, as opposed to an observational link (*i.e.*, A_c is not constrained to come after A_p) all actions A_s whose primary preconditions are supported by A_c are required to follow A_p (see Figure 5(a)). If $A_p \stackrel{q,e}{\rightarrow} A_c$ turns out to be observation link, no harm was done in adding the constraint $A_p \prec A_s$. The constraint is redundant, since A_s must follow A_p, by transitivity of the ordering relation ($A_p \prec A_c \prec A_s$).

Furthermore, if the effect e of A_p itself has a precondition supported by a (possibly later) action $A_{p'}$, the constraint $A_{p'} \prec A_s$ will also be added (see Figure 5(b)). In general, we require an action A_s to follow all other actions that provide support to one of its primary preconditions, where an action provides support to a given precondition if it directly supports the precondition or if it provides support to the precondition of an effect that directly supports the given precondition. These ordering constraints are added in the AddLink procedure (Figure 7).

When it comes time to execute an action, all causal effects whose preconditions are unknown, including assumptions, must be asserted as unknown in the agent's knowledge base. Additionally, all effects whose assumed preconditions have been verified should be asserted as true (this is valid, since the link guarantees that the agent didn't change the condition in the interim).

Evaluation

While the examples we have given in this paper show the benefits of a "leap before you look" approach, the real test is how well this approach actually works in a real planning domain. In fact, the evolution of the PUCCINI planner was driven by representational problems we encountered in trying to encode Unix and Internet action schemas for the Softbot, and trying to get a planner to produce reasonable plans using

HandleGoal($\mathcal{A}$, $\mathcal{O}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{G}$, $\mathcal{D}$)
if $\mathcal{G} \neq \emptyset$ then pop $\langle$g, $S_c\rangle$ from from $\mathcal{G}$ and select case:

1. If g = $\langle$Context $\Rightarrow$ *cond*$\rangle$, and Context$\models$*cond* then g is trivially satisfied.
2. If g is a **hands-off** goal, then call AddLink(A_0, g, nil, S_c, $\mathcal{O}$, $\mathcal{B}$)
3. **Else nondeterministically choose**
   (a) Reduce(g, $\mathcal{G}$)
   (b) Instantiate a new action A_{new} from $\mathcal{D}$, such that Satisfies(e, g) and add it to $\mathcal{A}$. Call Addlink(A_{new}, g, e, S_c, $\mathcal{O}$, $\mathcal{B}$). Add preconditions of A_{new} to $\mathcal{G}$.
   (c) Choose an existing action A_{old} from $\mathcal{A}$, such that Satisfies(e, g) and Call Addlink(S_p, g, e, A_{old}, $\mathcal{O}$, $\mathcal{B}$).
4. Propagate context labels

Figure 6: Procedure HandleGoal. The lines of the algorithm relevant to this paper are shown in **bold**.

Addlink(S_p, goal, eff, S_c, $\mathcal{O}$, $\mathcal{B}$)
(goal = $\langle$Context $\Rightarrow$ $g\rangle$; eff = **when** (p) e)

- If g is an **initially** goal, add $A_0 \overset{e,goal}{\rightarrow} S_p$ to $\mathcal{C}$. Otherwise, add $S_p \overset{e,goal}{\rightarrow} S_c$ to $\mathcal{C}$.
- **Unless g is an unannotated secondary precondition and e is an observe effect**
  1. **Add $S_p \prec S_c$ to $\mathcal{O}$**
  2. **If e is supported (directly or indirectly) by a potential verification link, whose producer is A_p, add $A_p \prec S_c$ to $\mathcal{O}$**
- Add MGU(e, g) to $\mathcal{B}$.
- Add $\langle\langle$Context $\Rightarrow$ $p\rangle$, $S_p\rangle$ to $\mathcal{G}$

Figure 7: Procedure Addlink. The lines of the algorithm relevant to this paper are shown in **bold**.

these schemas. Many of these struggles are discussed in (Golden & Weld 1996; Etzioni, Golden, & Weld 1997). One of the greatest representational gains came from the elimination of knowledge preconditions and the introduction of verification links. For example, when knowledge preconditions were used to encode actions, we needed no fewer than six encodings of the `finger` action (Figure 2), and even these six were not enough to fully capture the functionality of the one `finger` action we have now. Needless to say, this proliferation of actions presented greater search control problems, and the addition of more search control made the entire system more brittle.

Despite these gains, there are some potential pitfalls, which we should be on guard for. As we mentioned earlier, verification links can increase the size of the search space by giving the planner more ordering options. To determine whether this is a problem in practice, we ran three versions of PUCCINI on 10 representative Softbot goals. These goals are described in detail in Section 7.1.1 of (Golden 1997), but, briefly, they involve finding web pages, phone numbers and files (locally and via FTP) and compiling (LaTeX), displaying, printing, compressing and changing permissions on files. For example, goal #5 is "Display all web pages referenced by hyperlinks from both Dan Weld's homepage and Oren Etzioni's homepage," which requires finding the appropriate home pages, scanning both pages to find links in common, and then running Netscape on each common link.

Table 1 shows the statistics for solving these goals, using each version of PUCCINI. The three versions are as follows:

- **VL** is the version of PUCCINI presented here, which supports verification links and allows the producer of a verification link to follow the consumer.
- **NO** does not allow the producer to follow the consumer, but still allows self-links (*i.e.*, the producer is the consumer).
- **¬VL** disallows all verification links.

The statistics shown include both planning CPU time and real time for planning and execution. The real time reflects the time required to actually execute the commands and wait for completion, and thus represents the time that the user is most likely to care about. In the experiments we report, the Softbot easily solved the goals, using very little domain-dependent search control and executing the minimal number of actions needed to achieve the goals.

More importantly, we find that, for the goals that are solvable without verification links, the use of verification links has virtually no impact on the size of the search space. With the exception of goal 5 (for version ¬VL) the number of plans searched does not vary with the planner configuration. However, the number of solvable goals decreases significantly when verification links are disabled.

If verification links are entirely disabled (¬VL), only two of the goals are solvable. The reason for this is that the SADL encodings of actions like `ls` and `finger` are almost impossible to plan with if the planner doesn't support self-links. Due to this limitation, the comparison between VL and ¬VL is not entirely fair. In order to more fairly judge the impact of verification links, we ran the same problems on XII, the predecessor of PUCCINI. Since XII doesn't support verification links, the action encodings for that planner don't rely on them. Three of the goals, which rely on actions that were not part of the original XII domain theory, could not be solved by XII and were omitted from the test suite.

Table 2 shows the results for the seven remaining goals. Since the domain theories used by PUCCINI and XII are different, these performance results should be taken with a grain of salt, but they are suggestive. Of these goals, 1 and 2 are impossible to solve because the goals require the temporal expressiveness provided by the **initially** annotation, which XII does not support. Goal 9 is impossible to achieve without the use of verification links, despite the fact that the XII domain was

#	plans	exec	CPU (s)	real (s)
1	*	*	*	*
2	*	*	*	*
3	30	1	0.57	1.12
4	187	8	11.99	43.92
7	3619	11	539.59	602.83
9	*	*	*	*
10	792	6	65.92	87.06

Table 2: Planner statistics for seven out of ten sample goals, given to the XII planner, running on a Sun SPARCstation 20 "*" indicates that the goal is impossible for XII to achieve. Row and column labels are from Table 1

engineered to get around their absence. This goal is quite simple: Produce a color printout of a document and report the status of the print job. However, it cuts to the heart of a problem that stumped the Softbot team since the very beginning: Print jobs are produced by the `lpr` command, but `lpr` tells us nothing about them. To find out whether the print job actually exists and what identifier it has, we must execute `lpq`. Thus, the effect of `lpr`, the creation of a print job, is contingent on the job being sent to the print queue, a fact that can only be verified (by `lpq`) after the `lpr` has been executed. This does not present a problem if the planner supports verification links, but it creates a representational headache without them.

Conclusions

Past work in planning required agents to know, before executing an action, that the action would have its intended effect. We have shown that this restriction can be harmful, and we have shown that a simple change to a causal link planner, relaxing an ordering constraint, gives an agent the flexibility to subgoal on obtaining this knowledge when doing so would be fruitful, but also allows it to assume preconditions are true and later *verify* them to be true. We have shown that this mechanism can be implemented without impairing tractability.

Related Work

PUCCINI is an extension of XII (Golden, Etzioni, & Weld 1994), which is based on the UCPOP algorithm (Penberthy & Weld 1992). PUCCINI builds on XII by supporting a more expressive language, SADL (Golden & Weld 1996), and handling verification links. XII builds on UCPOP by dealing with information goals and effects, interleaving planning with execution and reasoning with Local Closed World knowledge (LCW) (Etzioni, Golden, & Weld 1997). The algorithm currently used for interleaving planning with execution builds on the approach used in IPEM (Ambros-Ingerson & Steel 1988). Unlike IPEM, PUCCINI can represent information goals as distinct from satisfaction goals. IPEM makes no such distinction, and thus cannot plan for information goals. PUCCINI also has its roots in the SOCRATES planner (Lesh 1992). Like PUCCINI, SOCRATES utilized the Softbot domain as its testbed and interleaved planning with execution. However, SOCRATES utilized knowledge preconditions and supported a less expressive action language (Etzioni *et al.* 1992).

We believe that our use of verification links is unique. However, it should be possible for a Partially Observable Markov Decision Process (POMDP) (Koenig 1992; Dean *et al.* 1995), or the planner C-BURIDAN (Draper, Hanks, & Weld 1994), to produce plans similar to those produced by PUCCINI using verification links. However, they accomplish this by following a generate-and-test approach: considering the addition of each possible sensor and testing the plan by checking it against a probability distribution on all possible worlds. They can also decide not to support the precondition of a conditional effect, provided the probability of that effect occurring anyway is sufficiently high. Using this approach, they can consider all plans that achieve the goal with a given probability, but the computational cost is daunting.

Some planners represent uncertain outcomes using conditional effects, and can execute actions for their uncertain effects (Kushmerick, Hanks, & Weld 1995; Draper, Hanks, & Weld 1994; Pryor & Collins 1996; Goldman & Boddy 1994). For example, Cassandra (Pryor & Collins 1996) represents uncertain outcomes as conditional effects with "`:unknown`" preconditions, and is capable of using these actions for their uncertain effects. Cassandra plans to achieve the goal for all possible outcomes of each action, and adds sensing actions to determine which outcome actually occurred. However, since Cassandra doesn't know what the actual preconditions of these effects are, it cannot subgoal on finding out whether the preconditions were true, after the fact. Furthermore, conditional effects without `:unknown` preconditions are treated in the usual way; the planner is forced to achieve the precondition if it wants the effect to occur.

PUCCINI can represent effects that are explicitly uncertain, by using the `U` truth value (Golden & Weld 1996), but, unlike Cassandra or C-BURIDAN, it can't execute these actions for their (uncertain) effects. This limitation stems from the fact that PUCCINI was not designed for contingency planning. In future extensions of PUCCINI, we would like to address this limitation.

Future Work

Whenever the planner makes a decision to verify a precondition after the fact, that introduces an uncertain outcome upon which the success of the plan depends. We refer to this as a source of *contingency* in the plan. There is always the possibility that the precondition is false, in which case the action won't have the desired effect. For example, if the agent is looking for the file `aips.tex`, and the planner adds `ls /papers` into the plan, there's a chance `aips.tex` will turn out not to be in `/papers`, in which case the plan will fail. In SADL, all sources of contingency stem from possibly unsatisfied preconditions (a precondition may be as simple as an equality constraint). The approach currently taken

prob	plans considered			actions executed			planning CPU (s)			real time (s)		
num	VL	NO	¬VL	VL	NO	¬VL	VL	NO	¬VL	VL	NO	¬VL
1	53	53	*	3	3	*	1.31	1.19	*	2.08	2.65	*
2	40	40	*	3	3	*	0.53	0.51	*	1.14	1.08	*
3	19	19	*	1	1	*	0.44	0.46	*	1.15	1.21	*
4	721	*	*	6	*	*	19.72	*	*	61.85	*	*
5	198	198	181	8	8	8	8.93	8.40	9.60	57.29	57.28	82.03
6	190	190	*	12	12	*	5.38	5.88	*	14.24	11.97	*
7	565	565	*	10	10	*	18.43	19.91	*	24.20	29.24	*
8	70	70	70	6	6	6	1.17	1.16	1.19	5.07	5.29	5.55
9	321	*	*	4	*	*	5.14	*	*	8.57	*	*
10	797	797	*	6	6	*	14.04	15.92	*	29.24	35.06	*

Table 1: Planner statistics for ten sample goals, running on a Sun SPARCstation 20. The results are shown for the PUCCINI planner in three configurations: with verification links and flexible ordering (VL), without flexible ordering (NO), and without any verification links (¬VL). "*" indicates that the goal is impossible for the planner in question to achieve.

to deal with contingency is to interleave planning and execution. However, some of the techniques used in PUCCINI, such as the use of context labels, are borrowed from contingency planning. We are in the process of integrating these techniques more completely, to produce a hybrid interleaved-contingency planner.

Acknowledgements

This research was funded by Office of Naval Research Grants N00014-94-1-0060 and N00014-98-1-0147, by National Science Foundation Grant IRI-9303461, and by ARPA / Rome Labs grant F30602-95-1-0024. Thanks to David Smith, Ellen Spertus, Richard Washington, Dan Weld and the anonymous reviewers for helpful comments.

References

Ambros-Ingerson, J., and Steel, S. 1988. Integrating planning, execution, and monitoring. In *Proc. 7th Nat. Conf. AI*, 735–740.

Dean, T.; Kaelbling, L. P.; Kirman, J.; and Nicholson, A. 1995. Planning under time constraints in stochastic domains. *J. Artificial Intelligence* 76.

Draper, D.; Hanks, S.; and Weld, D. 1994. Probabilistic planning with information gathering and contingent execution. In *Proc. 2nd Intl. Conf. AI Planning Systems.*

Etzioni, O., and Weld, D. 1994. A softbot-based interface to the Internet. *C. ACM* 37(7):72–6.

Etzioni, O.; Hanks, S.; Weld, D.; Draper, D.; Lesh, N.; and Williamson, M. 1992. An approach to planning with incomplete information. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, 115–125.

Etzioni, O.; Golden, K.; and Weld, D. 1997. Sound and efficient closed-world reasoning for planning. *J. Artificial Intelligence* 89(1–2):113–148.

Feynman, R. P. 1985. *Surely You're Joking, Mr. Feynman.* New York: Bantam Books.

Golden, K., and Weld, D. 1996. Representing sensing actions: The middle ground revisited. In *Proc. 5th Int. Conf. Principles of Knowledge Representation and Reasoning*, 174–185.

Golden, K.; Etzioni, O.; and Weld, D. 1994. Omnipotence without omniscience: Sensor management in planning. In *Proc. 12th Nat. Conf. AI*, 1048–1054.

Golden, K. 1997. *Planning and Knowledge Representation for Softbots.* Ph.D. Dissertation, University of Washington. Available as UW CSE Tech Report 97-11-05.

Goldman, R. P., and Boddy, M. S. 1994. Representing Uncertainty in Simple Planners. In *Proc. 4th Int. Conf. Principles of Knowledge Representation and Reasoning.*

Koenig, S. 1992. Optimal probabilistic and decision-theoretic planning using Markovian decision theory. UCB/CSD 92/685, Berkeley.

Kushmerick, N.; Hanks, S.; and Weld, D. 1995. An algorithm for probabilistic planning. *J. Artificial Intelligence* 76:239–286.

Lesh, N. 1992. A planner for a UNIX softbot. Internal report.

McAllester, D., and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proc. 9th Nat. Conf. AI*, 634–639.

Moore, R. 1985. A Formal Theory of Knowledge and Action. In Hobbs, J., and Moore, R., eds., *Formal Theories of the Commonsense World.* Ablex.

Morgenstern, L. 1987. Knowledge preconditions for actions and plans. In *Proceedings of IJCAI-87*, 867–874.

Pednault, E. 1986. *Toward a Mathematical Theory of Plan Synthesis.* Ph.D. Dissertation, Stanford University.

Penberthy, J., and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Int. Conf. Principles of Knowledge Representation and Reasoning*, 103–114. See also `http://www.cs.washington.edu/research/projects/ai/www/ucpop.html`.

Penberthy, J. 1993. *Planning with Continuous Change.* Ph.D. Dissertation, University of Washington. Available as UW CSE Tech Report 93-12-01.

Pryor, L., and Collins, G. 1996. Planning for contingencies: A decision-based approach. *J. Artificial Intelligence Research.*

Tate, A. 1977. Generating project networks. In *Proc. 5th Int. Joint Conf. AI*, 888–893.

Weld, D., and Etzioni, O. 1994. The first law of robotics (a call to arms). In *Proc. 12th Nat. Conf. AI*, 1042–1047.